Is Math Curvy?
Is Math Fat?

Ashani Dasgupta

Cheenta Academy

ashani.dasgupta@cheenta.com

March 9, 2026

Abstract

This exposition investigates the intrinsic geometric structure of human-formalized mathematical knowledge by embedding the Lean 4 *Mathlib* library into non-Euclidean manifolds. Using a dataset comprising approximately $440,000$ theorems and millions of logical dependencies, we employ Hyperbolic Neural Networks (HNNs) and discrete Ollivier-Ricci curvature analysis to characterize the "shape" of the mathematical universe.

Our empirical results reveal a dual scale geometry: while the globally the graph of theorems is tree-like (Gromov-hyperbolic with $\delta = 1$), locally it exhibits dense clustering (Jaccard-overlap proxy $\kappa \approx 0.022$). This tree of cliques reveals an intrinsic structure of the world of formalized mathematics.

We demonstrate the utility of this geometric duality through one *toy example* of AI-Mathematics collaboration: *Manifold Extrapolation*, where automated conjectures are generated by extending the manifold into peripheral *leaf* nodes.

Introduction

I have always been curious about the shape of mathematics. Usually a mathematician proves new theorems by gluing other previously known theorems. In this exposition we imagine theorems, definitions and axioms in mathematics as nodes or vertices in a graph. Suppose theorem A and theorem B are two such nodes. If theorem A is used directly in the proof of theorem B we add an edge from node A to node B. This process produces a directed graph of theorems in Mathematics. At the moment, we disregard the directions and we study the properties of this graph M.

There exists a database of theorems and formalized proofs in mathematics. This library is known as Mathlib. The proofs are written in a programming language know as Lean. For the purpose of this exposition we have used this library.

Initially we produced the graph M using approximately $4,40,000$ theorems, axioms and definitions. The number of edges, concocted using the recipe mentioned earlier, shot up to 9 million! Clearly some of the vertices are attached to many many other vertices. For example the vertex Eq is connected with $2,73,547$ other vertices. In Lean, Eq is the fundamental definition of equality. Formally it is an inductive type defined as the smallest relation that is reflexive. Do not worry if this sentence does not mean anything to you. We won't be getting into a tutorial of Lean programming language in this exposition. Instead we wish to study the geometry of the intrinsic structure of this library.

In order to remove the noise created by these high degree vertices, we prune them off from the graph. In other words we removed the top 1% vertices (and the edges incident on them). Suppose this new graph is M'.

We wish to investigate this mathematical universe. In particular we ask the following questions and address them using the pruned graph M':

- **Global Topology:** Is M' globally Gromov hyperbolic? Our stochastic sampling reveals a hyperbolicity constant of $\delta = 1.00$. Given the library's scale, this remarkably low value suggests a dominant tree-like global structure.
- **Local Geometry:** What is the "neighborhood" shape of M'? Measurement of the Ollivier-Ricci Curvature yields a mean value of $\kappa \approx 0.022$. This positive curvature indicates that locally, the graph is not a tree but a collection of dense, spherical clusters.
- **Manifold Mapping:** Can we find a matching manifold for M'? The combination of global hyperbolicity and local spherical congestion suggests a *Poincaré Ball* (Hyperbolic space) is the most suitable embedding space to reconcile these dual geometries.
- **Synthesizing Knowledge:** Can we extrapolate the manifold to predict new theorems? By identifying "voids" near the graph's periphery, we aim to demonstrate how AI can assist in automated conjecturing.

Graph of Mathematics, Gromov Hyperbolicity Estimation

In the first step of this investigation we construct the graph of theorems, definitions and axioms. We use a bit of Lean 4 and Python code to achieve this construction. If one does not know Python, then large language models can be used to produce the code and run in Google Colab or locally in Vscode. I encourage students to initially plan math experiments in natural language (say English). Once the key idea is in place, it is usually not be hard to convert the plan into code and perform the rest of the processing with the help of large language models.

To construct the theorem-dependency graph M, we used a custom extraction script in Lean 4. The script interfaces with the Lean kernel to perform inspection of every constant in the `Mathlib` and `Std` namespaces. Unlike a high-level text search, this method captures the true logical dependencies hidden within the expression trees of formal proofs.

The following code demonstrates the core logic used to harvest approximately 9.3 million edges. We utilize the `foldConsts` meta-programming tool to recursively collect every constant used in both the **Type** (the statement) and the **Value** (the proof body) of a declaration.

```
/-- Helper function to extract all dependencies --/
def getDeps (info : ConstantInfo) : Array String :=
  let empty : NameSet := {}

  -- 1. Collect from the Statement (Type)
  let set := info.type.foldConsts empty (fun n acc => acc.insert n)

  -- 2. Collect from the Proof (Value)
  let set := match info.value? with
    | some v => v.foldConsts set (fun n acc => acc.insert n)
    | none => set

  -- 3. Convert to Array of Strings for JSON
  set.foldl (fun acc n => acc.push (toString n)) #[]
```

One challenge in exporting formal math to JSON is the presence of Lean-specific notation (e.g., backslashes and quotes) that can break standard parsers. In order to avoid this, the script includes a `sanitize` function to ensure cross-platform compatibility with the Python-based NetworkX and PyTorch libraries used in later stages of this research.

```
def sanitize (s : String) : String :=
  let s1 := s.replace "\\" "\\\\"
  let s2 := s1.replace "\"" "\\\""
  let s3 := s2.replace "\n" " "
  s3
```

The script concludes by iterating through the environment's constants, filtering out internal system noise, and streaming the resulting nodes and edges into *mathlib graph* (json file). This file serves as the raw input for our manifold curvature and hyperbolicity calculations.

Notice that the graph has too many edges. Some vertices such as Eq have very large degree as they are used in many theorems. As discussed in the introduction, we remove the top 1% of the vertices, and edges incident on them from M. We call the new graph M'. All the remaining analysis is conducted on M'.

I am a student of Geometric Group Theory. I am curious to know whether M' is Gromov Hyperbolic. The notion of hyperbolicity and curvature has a fascinating history. Mathematicians like Carl Friedrich Gauss explored the notion of curvature in great detail. In classical treatises mathematicians used a lot of calculus-styled computations to understand curvature. We do not use that approach. Instead we use a relatively new description of hyperbolicity or negative curvature due to the work of Gromov.

M' is a graph. We wish to think of it as a metric space. Suppose a, b are any two points in M'. Then the shortest number of edges needed to traverse from a to b can be declared as the *distance* from a to b. This is a standard way to convert a graph into a metric space. M' is might not be connected. If there is no path between a pair of nodes, we think of them as infinitely far away from each other.

We continue using M' as a metric space with d as the distance function. The *Gromov product* of $x, y \in M'$ with respect to $z \in M'$ is defined as $(x, y)_z := \frac{1}{2}(d(x, z) + d(y, z) - d(x, y))$. M' is called δ-hyperbolic if for any $x, y, z, w \in M'$,

$$(x, y)_w \geq \min\{(x, z)_w, (y, z)_w\} - \delta.$$

The smallest δ for which this is satisfied is known as the Gromov hyperbolicity constant of M'. It is possible that in a metric space, we won't be able to find δ that works for all quadruples. Then we say that M' is not Gromov hyperbolic. This is an issue in infinite graphs. Theoretically our graph is finite. Hence, theoretically speaking, if we choose a large enough δ, it should work for all quadruples. It would be interesting if we are able to find a small δ. All this is very imprecise. That is okay at the moment.

We do face a computational issue. There are $4,30,000+$ nodes even after pruning. Choosing 4 nodes at a time produces $\binom{430000}{4}$ choices. For each quadruple x, y, z, w, we need to first calculate the following six distances.

- $d(x, y)$
- $d(x, z)$
- $d(x, w)$
- $d(y, z)$
- $d(y, w)$
- $d(z, w)$

In order to compute distances, we need to find the shortest path between the pairs of points in the graph. This uses something like Dijkstra's algorithm. One estimate is that we need around 200 mili-seconds to do this calculation in a standard computer (given the graph of our size). This alone would take 10 million years! Clearly we cannot do this.

After calculating the six distances, we need to compute the following quantities:

- $d(x, w) + d(y, w) - d(x, y)$
- $d(x, w) + d(z, w) - d(x, z)$
- $d(y, w) + d(z, w) - d(y, z)$

Finally, for each quadruple we compute $\frac{(S_{\max} - S_{mid})}{2}$ that is, half the difference between the largest and the middle of the three sums. That is the estimation of δ for this quadruple. We need to collect all such δ-s from all such quadruples and obtain a small enough upper bound that works for all.

As we noticed earlier that this is computationally impossible to accomplish in our life time. Hence we estimate δ using a sample of the population. We use 1000 randomly chosen quadruples and compute the δ.

We implement a stochastic estimation script in Python using the `NetworkX` library. This script picks 1000 random quadruples and calculates the hyperbolicity constant for each to find a global upper bound.

```
import networkx as nx
import json
import random
import numpy as np

# 1. Load the Pruned Graph
with open("mathlib_graph.json", 'r') as f:
    data = json.load(f)

G = nx.Graph()
for entry in data:
    u = entry['name']
    for v in entry['edges']:
        G.add_edge(u, v)

nodes = list(G.nodes())

def get_delta_sample(sample_size=1000):
    deltas = []
    for i in range(sample_size):
        # Pick 4 random nodes (a quadruple)
        quad = random.sample(nodes, 4)
        a, b, c, d = quad

        try:
            # Calculate the 6 pairwise distances
            dists = {
                'ab': nx.shortest_path_length(G, a, b),
                'cd': nx.shortest_path_length(G, c, d),
                'ac': nx.shortest_path_length(G, a, c),
                'bd': nx.shortest_path_length(G, b, d),
                'ad': nx.shortest_path_length(G, a, d),
                'bc': nx.shortest_path_length(G, b, c)
            }

            # Form the three pairwise sums
            sums = sorted([
                dists['ab'] + dists['cd'],
```

```
                dists['ac'] + dists['bd'],
                dists['ad'] + dists['bc']
            ], reverse=True)

            # Compute delta for the quadruple: (Max Sum - Mid Sum) / 2
            delta = (sums[0] - sums[1]) / 2.0
            deltas.append(delta)

        except nx.NetworkXNoPath:
            continue # Skip disconnected nodes

    return deltas

# Execute sampling
results = get_delta_sample(1000)
print(f"Max sampled delta: {max(results)}")
print(f"Mean sampled delta: {np.mean(results)}")
```

The results of our stochastic sampling is kind of remarkable. We found a maximum hyperbolicity constant of $\delta = 1.00$. Notice the slight technical deviation in the script: it implements the Four-Point Condition ($S_{\max} - S_{\text{mid}}$), which effectively measures the "thinness" of the quadrilaterals within the graph. This value of $\delta = 1.00$ is quite small for a library containing nearly ten million edges. It leads us to believe that the global structure of Mathlib is nearly as "tree-like" as the Poincaré Disk (where δ is typically small and bounded). In fact, this finding suggests that the logical "skeleton" of formalized mathematics is nearly as hyperbolic as the classical hyperbolic plane.

Is M' Locally Fat?

The Olliver - Ricci curvature is used in graph theory to estimate local fatness. Here is quick primer on the technique. Suppose x and y are two adjacent vertices in M'. Assume that x has three neighbors x_1, x_2, y and y has 5 neighbors y_1, y_2, y_3, y_4, x. We first put masses m_x, m_y on each of the vertices. The

$$m_x(x) = 0.5, m_x(y) = m_x(x_1) = m_x(x_2) = \frac{0.5}{3}$$

$$m_y(y) = 0.5, m_y(y_1) = m_y(y_2) = m_y(y_3) = m_y(y_4) = m_y(x) = \frac{0.5}{5}$$

We can compute the distance between pairs of points in this set-up. If there is no short-cut between y_i's and x_i's the graph locally appears as follows.

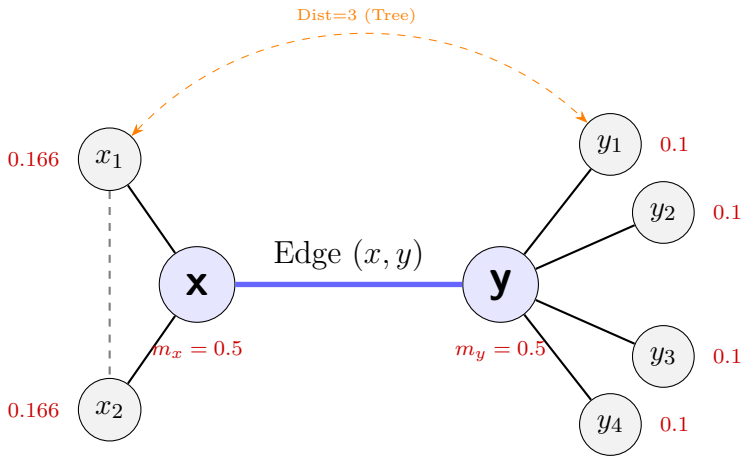

Figure 1: Local neighborhood of adjacent vertices x and y in M'. In a branching tree model, transport from x_1 to y_1 requires 3 steps, leading to negative curvature. In Mathlib, shortcut edges (triangles) often reduce this distance, resulting in the observed positive curvature $\kappa \approx 0.022$.

The *cost* of traveling from one vertex to another is the distance between those two vertices inside the graph. The following matrix makes that data explicit.

Distances $d(i,j)$	y	x	y_1	y_2	y_3	y_4
x	1	0	2	2	2	2
y	0	1	1	1	1	1
x_1	2	1	3	3	3	3
x_2	2	1	3	3	3	3

Table 1: The distance (cost) matrix for the neighborhoods of x and y assuming a branching tree structure. The entries represent the shortest path distance between nodes in the support of m_x (rows) and m_y (columns).

We now compute the Wasserstein distance $W_1(m_x, m_y)$ between x and y. It is essentially the minimal transport cost of transferring all the masses from x and its neighbors to y and its neighbors. The amount of mass a particular neighbor of y can accept is denoted by $m_y(*)$.

In our specific example we can do the following calculations. We look for the cheapest pairings. The plan looks like this:

1. Stationary Mass ($d = 0$):

- x (in m_x) sends 0.1 mass to x (in m_y). Cost: $0.1 \times 0 = 0$.
- y (in m_x) sends 0.166 mass to y (in m_y). Cost: $0.166 \times 0 = 0$.

2. Immediate Shifts ($d = 1$): The remaining 0.4 mass at x must go to y to fill the rest of y 's 0.5 capacity. Cost $0.4 \times 1 = 0.4$.

3. Neighbor Transport ($d \geq 2$) :

   - Now x and y are "full". The mass at x_1 and x_2 (0.332 total) must now travel to y_1, y_2, y_3, and y_4.
   - Distance $x_1 \to y_j = 3$ (in a tree). Cost: $0.332 \times 3 = 0.996$.

Total W_1 (Estimate): $0+0+0.4+0.996 = 1.396$. The Olliver - Ricci curvature is defined as follows:

$$\kappa(x, y) = 1 - \frac{W_1}{d(x, y)}$$

Hence in this case its value is:

$$\kappa = 1 - \frac{1.396}{1} = -0.396$$

This confirms that in a tree, the curvature is negative because the neighborhoods are spread too far apart.

Again, an exhaustive edge-by-edge computation over M', which contains more than nine million connections, would be prohibitively expensive. Instead of true Ollivier-Ricci curvature, we therefore used a Jaccard-overlap proxy on 20,000 randomly selected edges to estimate the local manifold signature. This stochastic procedure gives a tractable way to probe the library's local neighborhood structure by measuring how strongly adjacent nodes share common neighbors. Under this proxy, we found a mean overlap score of approximately 0.022 .

```
def estimate_ricci_curvature(u, v):
    set_u = set(G.neighbors(u))
    set_v = set(G.neighbors(v))

    intersection = len(set_u.intersection(set_v))
    union = len(set_u.union(set_v))

    # Estimate curvature based on neighborhood overlap
```

```
    kappa = (intersection / union) - (1 / (len(set_u) + len(set_v)))
    return kappa
```

The resulting mean of $\kappa \approx 0.022$ confirms that locally the graph is positively curved. This is expected as well. After-all the library is supposed to have a high density of logical triangles and cliques within specific mathematical domains.

So far the measurements exhibit a fascinating structural duality within the Mathlib dependency graph. On one hand, the Gromov hyperbolicity ($\delta = 1.00$) suggests a global branching structure characteristic of a hyperbolic tree. On the other hand, the positive Ollivier-Ricci curvature ($\kappa \approx 0.022$) indicates that the local neighborhoods are *fat* and highly interconnected, a trait of spherical geometry.

We describe this duality as *Tree of Cliques*. In this architecture, mathematics is not a uniform web, but a collection of dense, highly organized communities (cliques). Within a specific mathematical domain, such as Linear Algebra or Group Theory, theorems are densely interlinked. When we are *inside* a topic, the logic is redundant and interconnected, meaning the cost of moving between related ideas is very low ($W_1 < 1$). However, as we move between disparate fields, these dense clusters are connected by thin, branching *bridges* of logical dependency. This macro-structure follows the exponential expansion of a tree, meaning that at a high level, the shortest path between a theorem in Topology and a theorem in Number Theory must travel through a common logical *ancestor*. This hybrid geometry provides a rigorous justification for our use of the Poincaré Ball for theorem embedding. Hyperbolic space is uniquely suited to represent this Tree of Cliques because its metric properties naturally accommodate both the high-density local clustering and the vast, branching distances of the global logical hierarchy.

Hyperbolic Neural Network

We wish to embed the graph of mathematics M' in the Poincare ball model of hyperbolic space. In two dimension, we may think of it as a disc with radius 1 centered at $(0, 0)$ equipped with a peculiar distance function d_{hyp}. The distance function is strange. Two points close to the center that appear near each other are actually near each other according to d_{hyp}. But two visually similar distant points near the circumference are far - far away! We won't be discussing more on hyperbolic space in this exposition.

In particular we use a 16-dimensional ball and represent each node of M' as a point within this manifold. We train a hyperbolic neural network to find an *embedding*, a set of coordinates, for every theorem. The goal is to ensure that if two theorems are logically

related in Mathlib, their hyperbolic distance d_{hyp} in the ball is small, while unrelated theorems are pushed far apart. Once trained, the shortest logical path between any two mathematical concepts corresponds to the **geodesic** connecting their coordinates in the ball.

A neural network is an input-output machine. We may think of it as a function. Here is an example. Suppose we have a collection of 1000 grey-scale images of cats. Each image is 100×100 pixel. Each pixel is essentially a number between 0 and 255. We create a spreadsheet with 10001 columns. The first 10000 columns contain the pixel value of each image. In the 10001^{th} column the number 1 is written to confirm that the image corresponds to a cat. Now we *train* a model (that is build the function), with this data.

- Choose random coefficients $w_1, \cdots, w_{10000}$ known as weights and a random number b known as bias.
- Compute $z = w_1c_1 + w_2c_2 + \cdots + w_{10000}c_{10000} + b$. Here c_i are numbers in the i^{th} column.
- Inject a bit of non-linearity by using some non-linear function $y_{guess} = \phi(z) = \frac{1}{1+e^{-z}}$
- Check the difference y_{guess} with y (the number in the 10001^{th} column. This is known as *loss*.
- Use a bit of calculus to minimize the loss. To be more specific, we use the technique of gradient descent and check how infinitesimally changing each weight and bias, changes the loss. Accordingly we slightly dial down or dial up the value of each weight and bias. This is sometimes known as back-propagation.
- Once we do it with enough data, and enough number of times, we hope to minimize the loss and get ideal weights and biases.
- When a new image is used, we will use those ideal weights and biases on the pixel data, to decide whether it is a cat or not.

In a classical neural network, we change the old weights as follows:

$$\text{new weight} = \text{old weight} - \text{small step} \times \text{partial derivative of loss w.r.t weight}$$

In a hyperbolic neural network, we calibrate the steps according to the location of the point in the hyperbolic space. This is because, "0.1 units to the right" is different in

different locations on the hyperbolic space. In other words we perform a Riemannian Gradient Descent.

$$\text{new weight} = \exp_{\text{old position}}(\text{scaled gradient})$$

To map the high-dimensional logical structure of `mathlib` into the Poincaré Ball, we implement a Hyperbolic Neural Network using `PyTorch` and the `geoopt` library for Riemannian optimization. The model optimizes a contrastive loss to ensure that dependent theorems are close in hyperbolic distance while unrelated theorems are pushed apart.

```
[language=Python, caption=Training the Poincare Embedding with Riemannian Adam]
import torch
import torch.nn as nn
import geoopt
import json
import random

# Configuration
DIM = 16
CURVATURE = 1.0
LR = 0.3
BATCH_SIZE = 2048
NEG_SAMPLES = 5

class PoincareEmbedding(nn.Module):
    def __init__(self, num_nodes, dim):
        super().__init__()
        self.ball = geoopt.PoincareBall(c=CURVATURE)
        # Initialize nodes near the center of the ball
        init_data = self.ball.random(num_nodes, dim) * 0.01
        self.embeddings = geoopt.ManifoldParameter(init_data, manifold=self.ball)

    def forward(self, u_idx, v_idx):
        u = self.embeddings[u_idx]
        v = self.embeddings[v_idx]
        return self.ball.dist(u, v)

# Initialize Model and Riemannian Optimizer
model = PoincareEmbedding(len(names), DIM)
optimizer = geoopt.optim.RiemannianAdam(model.parameters(), lr=LR)
```

```
def train_step(u_pos, v_pos):
    # Real edges
    u_neg = u_pos.repeat_interleave(NEG_SAMPLES)
    v_neg = torch.randint(0, len(names), (len(u_pos) * NEG_SAMPLES,))

    optimizer.zero_grad()

    pos_dist = model(u_pos, v_pos)
    neg_dist = model(u_neg, v_neg)

    # Log-sigmoid loss to maximize distance gap
    pos_dist_expanded = pos_dist.repeat_interleave(NEG_SAMPLES)
    loss = -torch.log(torch.sigmoid(neg_dist - pos_dist_expanded) + 1e-6).mean()

    loss.backward()
    optimizer.step()
    return loss
```

The training of our hyperbolic neural network serves as the empirical bridge between discrete logical dependencies and continuous geometric space. We initialized the theorems near the origin ($\|x\| \approx 0.01$) and allowed the Riemannian Adam optimizer to sculpt their positions within the 16 -dimensional Poincaré Ball. Given the vast scale of M'-comprising over 9.3 million edges-we observed that a single training epoch provided sufficient variety in negative sampling to achieve convergence.

The loss function, which began at approximately 0.693 (the theoretical value for a random distribution), dropped sharply during the first two million edges before stabilizing at a final value of 0.182 . This significant reduction in loss confirms that the model successfully *gapped* the manifold, pulling logically dependent theorems together along hyperbolic geodesics while pushing unrelated concepts toward disparate sectors of the ball.

The fact that the loss plateaus at such a low value in a 16 -dimensional space is quite interesting to say the least. It seems to validates our earlier guess of the Tree of Cliques structure: the Poincaré Ball possesses the infinite room necessary to accommodate the exponential branching of mathematical knowledge ($\delta = 1.00$) without sacrificing the local density of mathematical communities ($\kappa \approx 0.022$). By the end of training, the model has effectively organized the library into a celestial hierarchy, where fundamental logic occupies the core and the leaves of modern research flare outward toward the infinite

boundary of the hyperbolic horizon.

Guessing new conjectures

We can now have some fun with our model. We will use it to guess conjectures. Here is a game-plan.

- Gather a few *peripheral* nodes that are near the circumference, near each other in the hyperbolic space, but possibly far away in the graph M'. We imagine that these are stars in the hyperbolic sky encircling a void.
- We wish to fill in the void with a virtual node.
- We use the peripheral node as input a large language model and produce a conjecture as an output. This conjecture is essentially guided by our hyperbolic neural network.

To identify the *logical voids*, we use the metric properties of the Poincaré Ball to calculate the Fréchet mean of peripheral clusters. Unlike Euclidean space, where an average is a simple arithmetic mean, the Fréchet mean in hyperbolic space accounts for the manifold's curvature, placing the *Virtual Node* at the precise point that minimizes the squared hyperbolic distances to all surrounding *stars*. This coordinate is then used to filter the local context for LLM-based conjecture synthesis.

We hope that M' might be extended infinitely using this strategy. In fact we may look for logical voids inside a particular domain. Particularly we search the domain of topology. Here is the code that we used.

```
import torch
import geoopt
import numpy as np

# --- CONFIGURATION ---
NAMESPACE_FILTER = "Topology"  # Change to "Algebra", "MeasureTheory", etc.
K_NEIGHBORS = 5                # Theorems surrounding the void

# 1. Load the trained HNN data
checkpoint = torch.load("mathlib_hnn_embeddings.pt")
embeddings = checkpoint['embeddings']
names      = checkpoint['names']
```

```
ball      = geoopt.PoincareBall(c=1.0)

# 2. Filter Nodes by Namespace (The Clique)
clique_indices = [i for i, name in enumerate(names)
                  if name.startswith(NAMESPACE_FILTER)]
clique_indices    = torch.tensor(clique_indices, dtype=torch.long)
clique_embeddings = embeddings[clique_indices]

# 3. Hunt for an Internal Void
start_idx   = np.random.choice(len(clique_indices))
anchor_node = clique_embeddings[start_idx]
distances   = ball.dist(anchor_node, clique_embeddings)
dist_values, local_indices = torch.topk(distances,
                                        k=K_NEIGHBORS,
                                        largest=False)

# 4. Collect cluster names for LLM prompt
cluster_names = []
for i in range(K_NEIGHBORS):
    node_idx = clique_indices[local_indices[i]].item()
    cluster_names.append(names[node_idx])
    print(f" - {names[node_idx]} (Dist: {dist_values[i].item():.4f})")
```

Surrounding Clique Theorems (Inputs for LLM)	**Hyperbolic Dist.**
`Topology.IsClosedEmbedding.isInducing`	0.0000
`Topology.IsLower.instClosedIciTopology`	6.1717
`Topology.lowerSet`	6.2809
`Topology.IsInducing.continuousConstSMul`	6.3270
`Topology.RelCWComplex.Subcomplex.instSetLike`	6.3307

Table 2: Peripheral nodes surrounding the conjectured void, identified by the Hyperbolic Neural Network. Distances are measured in the 16-dimensional Poincaré Ball.

Next we supply these ingredients to a large language model to produce the virtual node. Here is the output.

Theorem 1. *For a closed, inducing embedding of a convex lower set A into an ordered topological space with a continuous scalar multiplication, the subspace topology, the restricted order topology, and the topology inherited via the action all coincide.*

The two experiments in this exposition, taken together, suggest something quietly remarkable: the structure of formalized mathematics is neither a flat web nor a random tangle, but a tree of cliques. It is a hierarchical skeleton of branching logical ancestry, fleshed out locally by dense communities of interrelated ideas. The Poincaré Ball, with its infinite room near the boundary and its crowded center, turns out to be a natural home for this structure. The void-filling experiment is of course a toy example. The conjectures produced are rough, and their mathematical worth remains to be evaluated by experts. But the geometry is doing real work: it is selecting candidates that a flat keyword search would never find, precisely because their proximity is encoded in the shape of the library rather than its surface syntax. As a next step one may search for voids systematically across every major domain in Mathlib; Algebra, Analysis, Number Theory. We may build a conjecture dashboard. We could also ask the inverse question: given a known theorem that was hard to prove, does its position in the Poincaré Ball reveal why it was hard? Is it because it sat in a genuine void, far from any existing clique? We leave these questions open, and hope that students reading this will find them exciting.